Oracle9*i* SQLJ Developer's Guide and Reference
Release 1 (9.0.1)
Part Number A90212-01

1 Overview

This chapter provides a general overview of SQLJ features and scenarios. The following topics are discussed:

Introduction to SQLJ

This section introduces the basic concepts of SQLJ and discusses the complementary relationship between Java and PL/SQL in Oracle applications.

Basic Concepts

SQLJ enables applications programmers to embed SQL operations in Java code in a way that is compatible with the Java design philosophy. A SQLJ program is a Java program containing embedded SQL statements that comply with the ISO-standard SQLJ Language Reference syntax. Oracle9*i* SQLJ supports the SQLJ ISO standard specification. The standard covers only *static SQL* operations--those that are predefined and do not change in real-time as a user runs the application (although the data values that are transmitted can change dynamically). Oracle SQLJ also offers extensions to support *dynamic SQL* operations--those that are *not* predefined, where the operations themselves can change in real-time. (It is also possible to use dynamic SQL operations through JDBC code or PL/SQL code within a SQLJ application.) Typical applications contain much more static SQL than dynamic SQL.

SQLJ consists of both a translator and a runtime component and is smoothly integrated into your development environment. The developer runs the translator, with translation, compilation, and customization taking place in a single step when the `sqlj` front-end utility is run. The translation process replaces embedded SQL with calls to the SQLJ runtime, which implements the SQL operations. In

standard SQLJ this is typically, but not necessarily, performed through calls to a JDBC driver. To access an Oracle database, you would typically use an Oracle JDBC driver. When the end user runs the SQLJ application, the runtime is invoked to handle the SQL operations.

The Oracle SQLJ translator is conceptually similar to other Oracle precompilers and allows the developer to check SQL syntax, verify SQL operations against what is available in the schema, and check the compatibility of Java types with corresponding database types. In this way, errors can be caught by the developer instead of by a user at runtime. The translator checks the following:

- syntax of the embedded SQL
- SQL constructs, against a specified database schema to ensure consistency within a particular set of SQL entities (optional)

  It verifies table names and column names, for example.
- datatypes, to ensure that the data exchanged between Java and SQL have compatible types and proper type conversions

The SQLJ methodology of embedding SQL operations directly in Java code is much more convenient and concise than the JDBC methodology. In this way, SQLJ reduces development and maintenance costs in Java programs that require database connectivity.

Java and SQLJ versus PL/SQL

Java (including SQLJ) in Oracle applications does not replace PL/SQL. Java and PL/SQL are complementary to each other in the needs they serve.

While PL/SQL and Java can both be used to build database applications, the two languages were designed with different intents and, as a result, are suited for different kinds of applications:

- PL/SQL is a better solution for SQL-intensive applications. PL/SQL is optimized for SQL, and so SQL operations are faster in PL/SQL than in Java. Also, PL/SQL uses SQL datatypes directly, while Java applications must convert between SQL datatypes and Java types.
- Java, with its superior programming model, is a better solution for logic-intensive applications. Furthermore, Java's more general type system is better suited than PL/SQL for component-oriented applications.

Oracle provides easy interoperability between PL/SQL and Java, ensuring that you can take advantage of the strengths of both languages. PL/SQL programs can transparently call Java stored procedures, enabling you to build component-based Enterprise JavaBeans and CORBA applications. PL/SQL programs can have transparent access to a wide variety of existing Java class libraries through trivial PL/SQL call specifications.

Java programs can call PL/SQL stored procedures and anonymous blocks through JDBC or SQLJ. In particular, SQLJ provides syntax for calling stored procedures and functions from within a SQLJ statement, and also supports embedded PL/SQL anonymous blocks within a SQLJ statement.

Note:

Using PL/SQL anonymous blocks within SQLJ statements is one way to support dynamic SQL in a SQLJ application. (See "Dynamic SQL--DynamicDemo.sqlj" for a sample.) However, Oracle9*i* SQLJ includes extensions to support dynamic SQL directly. (See "Support for Dynamic SQL".

Overview of SQLJ Components

This section introduces the main SQLJ components and the concept of SQLJ *profiles*.

SQLJ Translator and SQLJ Runtime

Oracle SQLJ consists of two major components:

- Oracle SQLJ **translator**--This component is a precompiler that developers run after creating SQLJ source code.

  The translator, written in pure Java, supports a programming syntax that allows you to embed SQL operations inside SQLJ executable statements. SQLJ executable statements, as well as SQLJ declarations, are preceded by the `#sql` token and can be interspersed with Java statements in a SQLJ source code file. SQLJ source code file names must have the `.sqlj` extension. Here is a sample SQLJ statement:

```
#sql { INSERT INTO emp (ename, sal) VALUES ('Joe', 43000) };
```

  The translator produces a `.java` file and, for standard SQLJ code generation, one or more SQLJ *profiles*, which contain information about your SQL operations. SQLJ then automatically invokes a Java compiler to produce `.class` files from the `.java` file.

  Note:

  Alternatively, there is an Oracle-specific code generation setting that results in translation directly into Oracle JDBC code. In this case, no profiles are produced. See "Oracle-Specific Code Generation (No Profiles)".

- Oracle SQLJ **runtime**--Assuming standard SQLJ code generation, this component is invoked automatically each time an end user runs a SQLJ application.

  The SQLJ runtime, also written in pure Java, implements the desired actions of your SQL operations, accessing the database using a JDBC driver. The generic SQLJ standard does not require that a SQLJ runtime use a JDBC driver to access the database; however, Oracle SQLJ does require a JDBC driver, and, in fact, requires an Oracle JDBC driver if your application is customized with the default Oracle customizer (see below), or if Oracle-specific code generation is used.

For more information about the runtime, see "SQLJ Runtime".

In addition to the translator and runtime, there is a component known as the **customizer**. A customizer tailors your SQLJ profiles (if any) for a particular database implementation and vendor-specific features and datatypes. By default, the Oracle SQLJ front end invokes an Oracle customizer to tailor your profiles for an Oracle database and Oracle-specific features and datatypes.

When you use the Oracle customizer during translation, your application will require the Oracle SQLJ runtime and an Oracle JDBC driver when it runs.

SQLJ Profiles

With standard SQLJ code generation, SQLJ *profiles* are serialized Java resources (or, optionally, classes) generated by the SQLJ translator, which contain details about the embedded SQL operations in your SQLJ source code. The translator creates these profiles, then either serializes them and puts them into binary resource files, or puts them into `.class` files (according to your translator option settings).

Note:

As an alternative, Oracle SQLJ supports a setting for Oracle-specific code generation. In this case, the translator generates Oracle JDBC calls directly, and details of your embedded SQL operations are embodied in the JDBC calls. There are no profiles in this case. See "Oracle-Specific Code Generation (No Profiles)".

Overview of Profiles

SQLJ profiles are used (assuming standard SQLJ code generation) in implementing the embedded SQL operations in your SQLJ executable statements. Profiles contain information about your SQL operations and the types and modes of data being accessed. A profile consists of a collection of entries, where each entry maps to one SQL operation. Each entry fully specifies the corresponding SQL operation, describing each of the parameters used in executing this instruction.

SQLJ generates a profile for each connection context class in your application, where, typically, each connection context class corresponds to a particular set of SQL entities you use in your database operations. (There is one default connection context class, and you can declare additional classes.) The SQLJ standard requires that the profiles be of standard format and content. Therefore, for your application to use vendor-specific extended features, your profiles must be customized. By default, this occurs automatically, with your profiles being customized to use Oracle-specific extended features.

Profile customization allows vendors to add value in two ways:

- Vendors can support their own specific datatypes and SQL syntax. (For example, the Oracle customizer maps standard JDBC `PreparedStatement` method calls in translated SQLJ code to `OraclePreparedStatement` method calls, which provide support for Oracle type extensions.)
- Vendors can improve performance through specific optimizations.

For example, you must customize your profile to use Oracle objects in your SQLJ application.

Notes:

- By default, SQLJ profile file names end in the `.ser` extension, but this does not mean that all `.ser` files are profiles. Any serialized object uses that extension, and a SQLJ program unit can use serialized objects other than its profiles. (Optionally, profiles can be converted to `.class` files instead of `.ser` files.)
- A SQLJ profile is not produced if there are no SQLJ executable statements in the source code.

Binary Portability

SQLJ-generated profile files feature *binary portability*. That is, you can port them as is and use them with other kinds of databases or in other environments if you have not employed vendor-specific datatypes or features. This is true of generated `.class` files as well.

Overview of Oracle Extensions to the SQLJ Standard

With Oracle9*i* (and in Oracle8*i* release 8.1.7), Oracle SQLJ supports the SQLJ ISO specification. Because the SQLJ ISO standard is a superset of the SQLJ ANSI standard, it requires a JDK 1.2 or later environment that complies with J2EE. The SQLJ ANSI standard requires only JDK 1.1.x. The Oracle SQLJ translator accepts a broader range of SQL syntax than the ANSI SQLJ Standard specifies.

The ANSI standard addresses only the SQL92 dialect of SQL, but allows extension beyond that. Oracle SQLJ supports Oracle's SQL dialect, which is a superset of SQL92. If you need to create SQLJ programs that work with other DBMS vendors, avoid using SQL syntax and SQL types that are not in the standard and, therefore, may not be supported in other environments. (On your product CD, the directory `[Oracle Home]/sqlj/demo/components` includes a semantics-checker that you can use to verify that your SQLJ statements contain only standard SQL.)

For general information about Oracle SQLJ extensions, see Chapter 5, "Type Support", and Chapter 6, "Objects and Collections".

Oracle SQLJ Type Extensions

Oracle SQLJ supports the Java types listed below as extensions to the SQLJ standard. Do not use these or other types if you may want to use your code in other environments. To ensure that your application is portable, use the Oracle SQLJ `-warn=portable` flag. (See "Translator Warnings (-warn)".)

Using any of the following extensions requires Oracle customization or Oracle-specific code generation during translation, as well as the Oracle SQLJ runtime and an Oracle JDBC driver when your application runs.

- instances of `oracle.sql.*` classes as wrappers for SQL data (see "Support for JDBC 2.0 LOB

Types and Oracle Type Extensions")

- custom Java classes (classes that implement the `oracle.sql.ORAData` interface or the JDBC standard `java.sql.SQLdata` interface), typically produced by the Oracle JPublisher utility to correspond to SQL objects, object references, and collections (see "Custom Java Classes")

  Note, however, that the `SQLData` interface is standard. Classes that implement it are likely supported by other vendors' JDBC drivers and databases.

- stream instances (`AsciiStream`, `BinaryStream`, and `UnicodeStream`) used as output parameters (see "Support for Streams")

- iterator and result set instances as input or output parameters anywhere (the standard specifies them only in result expressions or cast statements; see "Using Iterators and Result Sets as Host Variables" and "Using Iterators and Result Sets as Stored Function Returns")

- Unicode character types--`NString`, `NCHAR`, `NCLOB`, `NcharAsciiStream`, and `NcharUnicodeStream` (see "Oracle SQLJ Extended Globalization Support")

Oracle SQLJ Functionality Extensions

In addition to the type extensions listed above, Oracle SQLJ supports the following extended functionality:

- Oracle-specific code generation

  This generates JDBC code directly. No profiles are produced and much of the SQLJ runtime functionality is bypassed during program execution. See "Oracle-Specific Code Generation (No Profiles)".

- dynamic SQL in SQLJ statements

  See "Support for Dynamic SQL".

- scrollable result set iterators with additional navigation methods, and `FETCH` syntax from result set iterators and scrollable result set iterators

  See "Scrollable Iterators".

Basic Translation Steps and Runtime Processing

This section introduces the following:

- basic steps of the Oracle SQLJ translator in translating SQLJ source code
- a summary of translator input and output
- runtime processing when a user runs your application

For more detailed information about the translation steps, see "Internal Translator Operations".

SQLJ source code contains a mixture of standard Java source together with SQLJ class declarations and SQLJ executable statements containing embedded SQL operations.

SQLJ source files have the `.sqlj` file name extension. The file name must be a legal Java identifier. If the source file declares a public class (maximum of one), then the file name must match the name of this class. If the source file does not declare a public class, then the file name should match the first defined class.

Translation Steps

After you have completed your `.sqlj` file, you must run SQLJ to process the files. (For coding the `.sqlj` file, basic SQLJ programming features and key considerations are discussed in Chapter 3 and Chapter 4.) The following example, for the source file `Foo.sqlj` whose first public class is `Foo`, shows SQLJ being run in its simplest form, with no command-line options:

```
sqlj Foo.sqlj
```

What this command actually runs is a front-end script or utility (depending on the platform) that reads the command line, invokes a Java virtual machine (JVM), and passes arguments to it. The JVM invokes the SQLJ translator and acts as a front end.

This document refers to running the front end as "running SQLJ" and to its command line as the "SQLJ command line". For information about command-line syntax, see "Command-Line Syntax and Operations".

From this point the following sequence of events occurs, presuming each step completes without fatal error.

1. The JVM invokes the SQLJ translator.

2. The translator parses the source code in the `.sqlj` file, checking for proper SQLJ syntax and looking for type mismatches between your declared SQL datatypes and corresponding Java host variables. (Host variables are local Java variables used as input or output parameters in your SQL operations. "Java Host Expressions, Context Expressions, and Result Expressions" describes them.)

3. The translator invokes the semantics-checker, which checks the syntax and semantics of embedded SQL statements. It also can optionally check the use of database elements in your code against an appropriate database schema.

   The developer can use online or offline checking, according to SQLJ option settings. If online checking is performed, then SQLJ will connect to a specified database schema to verify that the database supports all the database tables, stored procedures, and SQL syntax that the application uses, and that the host variable types in the SQLJ application are compatible with datatypes of corresponding database columns.

4. For standard SQLJ code generation, the translator processes your SQLJ source code, converts

SQL operations to SQLJ runtime calls, and generates Java output code and one or more SQLJ profiles. A separate profile is generated for each connection context class in your source code, where a different connection context class is typically used for each interrelated set of SQL entities that you use in your operations.

For Oracle-specific code generation, SQL operations are converted directly into Oracle JDBC calls and no profiles are produced. See "Oracle-Specific Code Generation (No Profiles)".

Generated Java code is put into a `.java` output file containing the following:

- any class definitions and Java code from your `.sqlj` source file
- class definitions created as a result of your SQLJ iterator and connection context declarations (see "Overview of SQLJ Declarations")
- a class definition for a specialized class (known as the *profile-keys* class) that SQLJ generates and uses in conjunction with your profiles (standard SQLJ code generation only)
- calls to the SQLJ runtime (for standard SQLJ code generation) or to Oracle JDBC (for Oracle-specific code generation) to implement the actions of your embedded SQL operations

  (The SQLJ runtime, in turn, uses the JDBC driver to access the database. See "SQLJ Runtime" for more information.)

Generated profiles (for standard SQLJ code generation only) contain information about all the embedded SQL statements in your SQLJ source code, such as actions to take, datatypes being manipulated, and tables being accessed. When your application is run, the SQLJ runtime accesses the profiles to retrieve your SQL operations and pass them to the JDBC driver.

By default, profiles are put into `.ser` serialized resource files, but SQLJ can optionally convert the `.ser` files to `.class` files as part of the translation.

5. The JVM invokes the Java compiler, which is usually, but not necessarily, the standard `javac` provided with the Sun Microsystems JDK.
6. The compiler compiles the Java source file generated in step 4 and produces Java `.class` files as appropriate. This will include a `.class` file for each class you defined, a `.class` file for each of your SQLJ declarations, and a `.class` file for the profile-keys class.
7. For standard SQLJ code generation, the JVM invokes the Oracle SQLJ customizer or other specified customizer.
8. For standard SQLJ code generation, the customizer customizes the profiles generated in step 4.

Notes:

- The preceding is a very generic example. It is also possible to specify pre-existing `.java` files on the command line to be compiled (and to be available

for type resolution as well), or to specify pre-existing profiles to be customized, or to specify `.jar` files containing profiles to be customized. See "Translator Command Line and Properties Files" for more information.

- SQLJ generates profiles and the profile-keys class only if your source code includes SQLJ executable statements, and only if you use standard SQLJ code generation.
- For standard SQLJ code generation, if you use the Oracle customizer during translation, your application will require the Oracle SQLJ runtime and an Oracle JDBC driver when it runs, even if your code does not use Oracle-specific features.
- For Oracle-specific code generation, your application will require an Oracle JDBC driver when it runs, even if your code does not use Oracle-specific features.

Summary of Translator Input and Output

This section summarizes what the SQLJ translator takes as input, what it produces as output, and where it puts its output.

Note:

This discussion mentions iterator class and connection context class declarations. Iterators are similar to JDBC result sets; connection contexts are used for database connections. For more information about these class declarations, see "Overview of SQLJ Declarations".

Input

In its most basic operation, the SQLJ translator takes one or more `.sqlj` source files as input in its command line. The name of your main `.sqlj` file is based on the public class it defines, if it defines one, or else on the first class it defines if there are no public class definitions. Each public class you define must be in its own `.sqlj` file.

If your main `.sqlj` file defines class `MyClass`, then the source file name must be:

```
MyClass.sqlj
```

This must also be the file name if there are no public class definitions but `MyClass` is the first class defined.

When you run SQLJ, you can also specify numerous SQLJ options in the command line or properties

files.

For more information about SQLJ input, including additional types of files you can specify in the command line, see "Translator Command Line and Properties Files".

Output

The translation step produces a Java source file for each `.sqlj` file in your application, and (with standard SQLJ code generation) at least one application profile (presuming your source code uses SQLJ executable statements).

SQLJ generates source files and profiles as follows:

- Java source files will be `.java` files with the same base names as your `.sqlj` files.

  For example, `MyClass.sqlj` defines class `MyClass` and the translator produces `MyClass.java`.

- The application profile files, if generated, contain information about the SQL operations of your SQLJ application. There will be one profile for each connection class that you use in your application. The profiles will have names with the same base name as your main `.sqlj` file, plus the following extensions:

  ```
  _SJProfile0.ser
  _SJProfile1.ser
  _SJProfile2.ser
  ...
  ```

  For example, for `MyClass.sqlj` the translator produces:

  ```
  MyClass_SJProfile0.ser
  ```

  The `.ser` file extension reflects the fact that the profiles are serialized. The `.ser` files are binary files.

  Note:

  There is a translator option, `-ser2class`, that instructs the translator to generate profiles as `.class` files instead of `.ser` files. Other than the file name extension, the naming is the same.

The compilation step compiles the Java source file into multiple class files. For standard SQLJ code generation there are at least two class files: one for each class you define in your `.sqlj` source file (minimum of one), and one for a class, known as the *profile-keys* class, that the translator generates and uses with the profiles to implement your SQL operations (presuming your source code uses SQLJ executable statements). Additional `.class` files are produced if you declared any SQLJ iterators or connection contexts (see "Overview of SQLJ Declarations"). Also, separate `.class` files will be produced for any inner classes or anonymous classes in your code. For Oracle-specific code generation,

no profiles or profile-keys class are produced. For information about Oracle-specific code generation, see "Oracle-Specific Code Generation (No Profiles)".

The `.class` files are named as follows:

- The class file for each class you define consists of the name of the class with the `.class` extension.

  For example: `MyClass.sqlj` defines `MyClass`, the translator produces the `MyClass.java` source file, and the compiler produces the `MyClass.class` class file.

- The profile-keys class that the translator generates is named according to the base name of your main `.sqlj` file, plus the following:

  ```
  _SJProfileKeys
  ```

  So the class file has the following extension:

  ```
  _SJProfileKeys.class
  ```

  For example, for `MyClass.sqlj`, the translator together with the compiler produce:

  ```
  MyClass_SJProfileKeys.class
  ```

- The translator names iterator classes and connection context classes according to how you declare them. For example, if you declare an iterator `MyIter`, there will be a `MyIter.class` class file.

The customization step alters the profiles but produces no additional output.

> **Note:**
>
> It is not necessary to reference SQLJ profiles or the profile-keys class directly. This is all handled automatically.

Output File Locations

By default, SQLJ places generated `.java` files in the same directory as your `.sqlj` file. You can specify a different `.java` file location, however, using the SQLJ `-dir` option.

By default, SQLJ places generated `.class` and `.ser` files in the same directory as the generated `.java` files. You can specify a different `.class` and `.ser` file location, however, using the SQLJ `-d` option. This option setting is passed to the Java compiler so that `.class` files and `.ser` files will be in the same location.

For either the `-d` or `-dir` option, you must specify a directory that already exists. For more information

about these options, see "Options for Output Files and Directories".

Runtime Processing

This section discusses runtime processing during program execution, considering both standard SQLJ code generation and Oracle-specific code generation.

Processing for Standard SQLJ Generated Code

When a user runs the application, the SQLJ runtime reads the profiles and creates "connected profiles", which incorporate database connections. Then the following occurs each time the application must access the database:

1. SQLJ-generated application code uses methods in a SQLJ-generated profile-keys class to access the connected profile and read the relevant SQL operations. There is a mapping between SQLJ executable statements in the application and SQL operations in the profile.

2. The SQLJ-generated application code calls the SQLJ runtime, which reads the SQL operations from the profile.

3. The SQLJ runtime calls the JDBC driver and passes the SQL operations to it.

4. The SQLJ runtime passes any input parameters to the JDBC driver.

5. The JDBC driver executes the SQL operations.

6. If any data is to be returned, the database sends it to the JDBC driver, which sends it to the SQLJ runtime for use by your application.

> **Note:**
>
> Passing input parameters (step 4) can also be referred to as "binding input parameters" or "binding host expressions". The terms *host variables*, *host expressions*, *bind variables*, and *bind expressions* are all used to describe Java variables or expressions that are used as input or output for SQL operations.

Processing for Oracle-Specific Generated Code

When you translate with the translator setting `-codegen=oracle`, your program at runtime will execute the following:

- Oracle-specific APIs in the SQLJ runtime that ensure batching support and proper creation and closing of Oracle JDBC statements

- direct calls into the Oracle JDBC APIs for registering, passing, and retrieving parameters and result sets

See "Oracle-Specific Code Generation (No Profiles)".

Alternative Deployment Scenarios

Although this manual mainly discusses writing for client-side SQLJ applications, you may find it useful to run SQLJ code in the following scenarios:

- from an applet
- in the server (optionally running the SQLJ translator in the server as well)
- against Oracle Lite

Running SQLJ in Applets

Because the SQLJ runtime is pure Java, you can use SQLJ source code in applets as well as applications. There are, however, a few considerations, as discussed below.

For an example, see "Applet Sample".

For applet issues that apply more generally to the Oracle JDBC drivers, see the *Oracle9i JDBC Developer's Guide and Reference*, which includes discussion of firewalls and security issues as well.

General Development and Deployment Considerations

The following general considerations apply to the use of Oracle SQLJ applets.

- You must package all the SQLJ runtime packages with your applet:

  ```
  sqlj.runtime
  sqlj.runtime.ref
  sqlj.runtime.profile
  sqlj.runtime.profile.ref
  sqlj.runtime.error
  ```

  as well as the following if you used Oracle customization:

  ```
  oracle.sqlj.runtime
  oracle.sqlj.runtime.error
  ```

  These classes are included with your Oracle installation in one of several runtime libraries in the `[Oracle Home]/lib` directory (see "Requirements for Using Oracle SQLJ").

- You must specify a pure Java JDBC driver, such as the Oracle JDBC Thin driver, for your database connection.
- You must explicitly specify a connection context instance for each SQLJ executable statement in an applet. This is a requirement because you could conceivably run two SQLJ applets in a single

browser and, thus, in the same JVM. (For information about connections, see "Connection Considerations".)

- You may want to use the translator setting `-codegen=oracle` to generate Oracle-specific code. This will eliminate the use of Java reflection at runtime, thereby increasing portability across different browser environments. For information about the `-codegen` option, see "Code Generation (-codegen)". For general information about Oracle-specific code generation, see "Oracle-Specific Code Generation (No Profiles)".

General End User Considerations

When end users run your SQLJ applet, classes in their classpath may conflict with classes that are downloaded with the applet.

Oracle, therefore, recommends that end users clear their classpath before running the applet.

Java Environment and the Java Plug-in

Here are some additional considerations regarding the Java environment and use of Oracle-specific features.

- SQLJ requires the runtime environment of JDK 1.1.x or higher. Users cannot run SQLJ applets in browsers employing JDK 1.0.x, such as Netscape Navigator 3.x and Microsoft Internet Explorer 3.x, without a plug-in or some other means of using JRE 1.1.x instead of the browser's default JRE.

  One option is to use a Java plug-in offered by Sun Microsystems. For information, refer to the following Web site:

  ```
  http://www.javasoft.com/products/plugin
  ```

- Some browsers, such as Netscape Navigator 4.x, do not support resource files with a `.ser` extension, which is the extension employed by the SQLJ serialized object files that are used for profiles. The Sun Microsystems Java plug-in, however, supports `.ser` files.

  Alternatively, if you do not want to use the plug-in, Oracle SQLJ offers the `-ser2class` option to convert `.ser` files to `.class` files during translation. See "Conversion of .ser File to .class File (-ser2class)" for more information.

  Note:

  These considerations do not apply to Oracle-specific code generation, where no profiles are produced.

- Applets using Oracle-specific features require the Oracle SQLJ runtime to work. The Oracle runtime consists of the classes in the SQLJ runtime library file under `oracle.sqlj.*`. The Oracle SQLJ runtime library requires the Java Reflection API (`java.lang.reflect.*`); the `runtime11`,

`runtime12`, and `runtime12ee` runtime libraries must use the Reflection API only in the circumstances outlined below. Most browsers do not support the Reflection API or impose security restrictions, but Sun's Java plug-in provides support for the Reflection API.

Note:

The term "Oracle-specific features" refers both to the use of Oracle type extensions (discussed in Chapter 5, "Type Support") and the use of SQLJ features that require your application to be customized to work against an Oracle database (for example, this is true of the `SET` statement, discussed in Chapter 3, "Basic Language Features").

With standard SQLJ code generation, the following SQLJ language features always require the Java Reflection API (`java.lang.reflect.*`), regardless of the version of the SQLJ runtime you are using:

- the `CAST` statement
- REF CURSOR parameters or REF CURSOR columns being retrieved from the database as instances of a SQLJ iterator
- retrieval of `java.sql.Ref`, `Struct`, `Blob`, or `Clob` objects
- retrieval of SQL objects as instances of Java classes implementing the `oracle.sql.ORAData` or `java.sql.SQLData` interfaces

---

Notes:

- An exception to the preceding is if you use SQLJ in a mode that is fully compatible with ISO. That is, if you use SQLJ in an environment that complies with J2EE and you translate and run your program with the SQLJ `runtime12ee` library, and you employ connection context type maps as specified by ISO. In this case, instances of `java.sql.Ref, Struct, Blob, Clob`, and `SQLData` are being retrieved without the use of reflection.
- Also, if you use Oracle-specific code generation (translator setting `-codegen=oracle`), you will eliminate the use of reflection in all of the instances listed above.

- Consider using the `runtime11` library for your applets. Doing so permits you to use Oracle-specific features and Oracle-specific customization.
- If your applet does not use any Oracle-specific features, you can distribute it with the generic

SQLJ runtime library, `runtime-nonoracle`. To support this, do not customize the applet during translation (for standard code generation) and do not use Oracle-specific code generation. Set `-profile=false` when you translate the code. (See "Profile Customization Flag (-profile)".) If you neglect to set `-profile=false`, then the default Oracle customizer will load Oracle-specific runtime classes. This will result in your applet requiring the Oracle runtime even though it does not use Oracle-specific features.

The preceding issues can be summarized as follows, focusing on users with Internet Explorer and Netscape browsers:

- Distribute your applet with the `runtime11` and `classes111` libraries. In this case, the SQLJ and JDBC versions must match. For example, to use the SQLJ 9.0.0 runtime, you must have the Oracle 9.0.0 JDBC driver.
- If you use object types, JDBC 2.0 types, REF CURSORs, or the `CAST` statement in your SQLJ statements, then you must adhere to your choice of the following:
  - The browser in which you run supports JDK 1.1 or higher and permits reflection.

  or:

  - You run your applet through a browser Java plug-in.

  or:

  - You use the `-codegen=oracle` setting when you translate your applet.
- If your applet does not use Oracle-specific features, then you can compile it without customization (`-profile=false`) and distribute it with the generic SQLJ runtime, `runtime-nonoracle`.

> **Note:**
>
> For an example of a generic SQLJ applet (not using Oracle-specific features), see "Applet Sample".

Introduction to SQLJ in the Server

In addition to its use in client applications, SQLJ code can run within a target Oracle9*i* database or middle-tier database cache in stored procedures, stored functions, triggers, Enterprise JavaBeans, or CORBA objects. Server-side access occurs through an Oracle JDBC driver that runs inside the server itself. Additionally, the Oracle9*i* database has an embedded SQLJ translator so that SQLJ source files for server-side use can optionally be translated directly in the server.

The two main areas to consider, which Chapter 11, "SQLJ in the Server", discusses in detail are:

- creating SQLJ code for use within the server

Coding a SQLJ application for use within the target Oracle9*i* database is similar to coding for client-side use. What issues do exist are due to general JDBC characteristics, as opposed to SQLJ-specific characteristics. The main differences involve connections:

- You have only one connection.
- The connection is to the database in which the code is running.
- The connection is implicit (does not have to be explicitly initialized, unlike on a client).
- The connection cannot be closed--any attempt to close it will be ignored.

Additionally, the JDBC server-side driver used for connections within the server does not support auto-commit mode.

> **Note:**
>
> There is also a server-side Thin driver for connecting to one server from code that runs in another. This case is effectively the same as using a Thin driver from a client and is coded in the same way. See "Overview of the Oracle JDBC Drivers".

- translating and loading SQLJ code for server-side use

You can translate and compile your code either on a client or in the server. If you do this on a client, you can then load the class and resource files into the server from your client machine, either pushing them from the client using the Oracle `loadjava` utility or pulling them in from the server using SQL commands. (It is convenient to have them all in a single `.jar` file first.)

Alternatively, you can translate and load in one step, using the embedded server-side SQLJ translator. If you load a SQLJ source file instead of class or resource files, then translation and compilation are done automatically. In general, `loadjava` or SQL commands can be used for class and resource files or for source files. From a user perspective `.sqlj` files are treated the same as `.java` files, with translation taking place implicitly.

See "Loading SQLJ Source and Translating in the Server" for information about using the embedded server-side translator.

> **Note:**
>
> The server-side translator does not support the Oracle SQLJ `-codegen` option. For Oracle-specific generated code to run in the server, you must translate on a client and load the individual components into the server. Also note restrictions on interoperability when running code generated with different settings. For more information, see "Translating SQLJ Source on a Client and Loading Components" and "Oracle-Specific Code Generation (No Profiles)".

Using SQLJ with Oracle Lite

You can use SQLJ on top of Oracle Lite. This section provides a brief overview of this functionality. For more information, refer to the *Oracle Lite Java Developer's Guide*.

Overview of Oracle Lite and Java Support

Oracle Lite is a lightweight database that offers flexibility and versatility that larger databases cannot. It requires only 350K to 750K of memory for full functionality, natively synchronizes with the Palm Computing platform, and can run on Windows NT (3.51 or higher), Windows 95, and Windows 98. It offers an embedded environment that requires no background or server processes.

Oracle Lite is compatible with Oracle9*i*, Oracle8*i*, Oracle8, and Oracle7. It provides comprehensive support for Java, including JDBC, SQLJ, and Java stored procedures. There are two alternatives for access to Oracle Lite from Java programs:

- native JDBC driver

  This is intended for Java applications that use the relational data model, allowing them direct communication with the object-relational database engine.

  Use the relational data model if your program has to access data that is already in SQL format, must run on top of other relational database systems, or uses very complex queries.

- Java Access Classes (JAC)

  This is intended for Java applications that use either the Java object model or the Oracle Lite object model, allowing them to access persistent information stored in Oracle Lite, without having to map between the object model and the relational model. Use of JAC also requires a persistent Java proxy class to model the Oracle Lite schema. This can be generated by Oracle Lite tools.

  Use the object model if you want your program to have a smaller footprint and run faster and you do not require the full capability of the SQL language.

There is interoperability between Oracle Lite JDBC and JAC, with JAC supporting all types that JDBC supports, and JDBC supporting JAC types that meet certain requirements.

Requirements to Run Java on Oracle Lite

Note the following requirements if you intend to run a Java program on top of Oracle Lite:

- Windows NT 3.51 or higher, Windows 95, or Windows 98
- Oracle Lite 3.0 or higher
- JDK 1.1.x or higher
- Java Runtime Environment (JRE) that supports Java Native Interface (JNI)

The JREs supplied with JDK 1.1.x and higher, Oracle JDeveloper, and Symantec Visual Cafe support JNI.

Support for Oracle Extensions

Oracle Lite 4.0.x. includes an Oracle-specific JDBC driver and Oracle-specific SQLJ runtime classes (including the Oracle semantics-checkers and customizer), allowing use of Oracle-specific features and type extensions.

Alternative Development Scenarios

The discussion in this book assumes that you are coding manually in a UNIX environment for English-language deployment. However, you can use SQLJ on other platforms and with IDEs. There is also globalization support for deployment to other languages. This section introduces these topics:

- globalization support
- SQLJ in IDEs
- Windows considerations

SQLJ Globalization Support

Oracle SQLJ support for native languages and character encodings is based on Java's built-in globalization support capabilities.

The standard `user.language` and `file.encoding` properties of the JVM determine appropriate language and encoding for translator and runtime messages. The SQLJ `-encoding` option determines encoding for interpreting and generating source files during translation.

For information, see "Globalization Support in the Translator and Runtime".

SQLJ in JDeveloper and Other IDEs

Oracle SQLJ includes a programmatic API so that it can be embedded in integrated development environments (IDEs) such as Oracle JDeveloper. The IDE takes on a role similar to that of the `sqlj` script used as a front end in Solaris, invoking the translator, semantics-checker, compiler, and customizer.

Oracle JDeveloper is a Windows NT-based visual development environment for Java programming. The JDeveloper Suite enables developers to build multi-tier, scalable Internet applications using Java across the Oracle Internet Platform. The core product of the suite--the JDeveloper Integrated Development Environment--excels in creating, debugging, and deploying component-based applications.

The Oracle JDBC OCI and Thin drivers are included with JDeveloper, as well as drivers to access Oracle Lite.

JDeveloper's compilation functionality includes an integrated Oracle SQLJ translator so that your SQLJ application is translated automatically as it is compiled.

Information about JDeveloper is available at the following URL:

```
http://technet.oracle.com
```

Windows Considerations

Note the following if you are using a Windows platform instead of Solaris:

- This manual uses Solaris/UNIX syntax. Use platform-specific file names and directory separators (such as "\" on Windows) that are appropriate for your platform, because your JVM expects file names and paths in the platform-specific format. This is true even if you are using a shell (such as ksh on NT) that permits a different file name syntax.

- For Solaris, Oracle SQLJ provides a front-end script, sqlj, that you use to invoke the SQLJ translator. On Windows, Oracle SQLJ instead provides an executable file, sqlj.exe. Using a script is not feasible on Windows platforms because .bat files on these platforms do not support embedded equals signs (=) in arguments, string operations on arguments, or wildcard characters in file name arguments.

- How to set environment variables is specific to the operating system. There may also be OS-specific restrictions. In Windows 95, use the Environment tab in the System control panel. Additionally, since Windows 95 does not support the "=" character in variable settings, SQLJ supports the use of "#" instead of "=" in setting SQLJ_OPTIONS, an environment variable that SQLJ can use for option settings. Consult your operating system documentation regarding settings and syntax for environment variables, and be aware of any size limitations.

- As with any operating system and environment you use, be aware of specific limitations. In particular, the complete, expanded SQLJ command line must not exceed the maximum command-line size, which is 250 characters for Windows 95 and 4000 characters for Windows NT. Consult your operating system documentation.

- On Windows, it is possible that the SQLJ translation process will suspend during compilation. If you encounter this problem, use the translator -passes option, which is discussed in "SQLJ Two-Pass Execution (-passes)".

Refer to the Windows platform README file for additional information.